# 一、Linux驱动分离与分层

1、以前的驱动实验都很简单，就是对IO的操作。

目的：方便驱动的编写，提高软件的重用以及跨平台性能。

## 1.1、驱动的分隔与分离

主机驱动，设备驱动

1、单片机驱动开发，IIC设备MPU6050，

2、将驱动分离：主机控制器驱动和设备驱动，主机控制器驱动一般是半导体厂商写的。在linux驱动框架下编写具体的设备驱动。

3、中间的联系就是核心层。

## 1.2、驱动分层

# 二、总线-驱动-设备

驱动-总线-设备。

根据驱动的分离与分层衍生出了总线(bus)-驱动(driver)-设备(device)驱动框架。

总线代码我们不需要编写，linux内核提供给我们使用的。我们需要编写驱动和设备，当向总线注册驱动的时候，总线会从现有的所有设备中查找，看看哪个设备和此驱动匹配。同理，当向总线注册设备的时候总线也会在现有的驱动中查看与之匹配的驱动。

驱动：是具体的设备驱动

设备：设备属性，包括地址范围、如果是IIC的话还有IIC器件地址、速度

## 2.1、总线

总线数据类型为：bus\_type。向内核注册总线使用bus\_register。

struct bus\_type {

const char \*name;

const char \*dev\_name;

struct device \*dev\_root;

struct device\_attribute \*dev\_attrs; /\* use dev\_groups instead \*/

const struct attribute\_group \*\*bus\_groups;

const struct attribute\_group \*\*dev\_groups;

const struct attribute\_group \*\*drv\_groups;

**int (\*match)(struct device \*dev, struct device\_driver \*drv);**

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

int (\*probe)(struct device \*dev);

int (\*remove)(struct device \*dev);

void (\*shutdown)(struct device \*dev);

int (\*online)(struct device \*dev);

int (\*offline)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

const struct iommu\_ops \*iommu\_ops;

struct subsys\_private \*p;

struct lock\_class\_key lock\_key;

};

向linux内核注册总线，使用bus\_register函数。bus\_unregister函数卸载。

1、总线主要工作就是完成总线下的设备和驱动之间的匹配。

## 2.2、驱动

驱动数据类型为device\_driver，驱动程序向内核注册驱动采用driver\_register。

struct device\_driver {

const char \*name;

**struct bus\_type \*bus;**

struct module \*owner;

const char \*mod\_name; /\* used for built-in modules \*/

bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/

**const struct of\_device\_id \*of\_match\_table;**

const struct acpi\_device\_id \*acpi\_match\_table;

**int (\*probe) (struct device \*dev);**

**int (\*remove) (struct device \*dev);**

**void (\*shutdown) (struct device \*dev);**

**int (\*suspend) (struct device \*dev, pm\_message\_t state);**

**int (\*resume) (struct device \*dev);**

const struct attribute\_group \*\*groups;

const struct dev\_pm\_ops \*pm;

struct driver\_private \*p;

};

驱动和设备匹配以后驱动里面的probe函数就会执行。

使用driver\_register注册驱动。

driver\_register

-> bus\_add\_driver

-> driver\_attach //查找bus下所有设备，找预期匹配的。

->bus\_for\_each\_dev(drv->bus, NULL, drv, \_\_driver\_attach);

->\_\_driver\_attach //每个设备都调用此函数，

//查看每个设备是否与驱动匹配

-> driver\_match\_device //检查是否匹配。

-> driver\_probe\_device

->really\_probe

-> drv->probe(dev); //执行driver的probe函数。

向总线注册驱动的时候，会检查当前总线下的所有设备，有没有与此驱动匹配的设备，如果有的话就执行驱动里面的probe函数。

## 2.2、设备

设备数据类型为device，通过device\_register向内核注册设备。

struct device {

struct device \*parent;

struct device\_private \*p;

struct kobject kobj;

const char \*init\_name; /\* initial name of the device \*/

const struct device\_type \*type;

struct mutex mutex; /\* mutex to synchronize calls to

\* its driver.

\*/

**struct bus\_type \*bus; /\* type of bus device is on \*/**

**struct device\_driver \*driver; /\* which driver has allocated this**

**device \*/**

……

};

向总线注册设备的时候，使用device\_register。

device\_register

-> device\_add

-> bus\_add\_device

-> bus\_probe\_device

->device\_attach

->bus\_for\_each\_drv(dev->bus, NULL, dev, \_\_device\_attach);

-> \_\_device\_attach

-> driver\_match\_device //匹配驱动

->bus->match

->driver\_probe\_device //执行此函数

-> really\_probe

**-> drv->probe(dev);**

驱动与设备匹配以后驱动的probe函数就会执行，probe函数就是驱动编写人员去编写的！！！！

# 三、platform平台驱动模型

根据总线-驱动-设备驱动模型，IIC、SPI、USB这样实实在在的总线是完全匹配的，但是要有一些外设是没法归结为具体的总线：比如定时器、RTC、LCD等。为此linux内核创造了一个虚拟的总线：platform总线。

1、方便开发，linux提出了驱动分离与分层。

2、进一步引出了驱动-总线-设备驱动模型，或者框架。

3、对于SOC内部的RTC，timer等等不好归结为具体的总线，为此linux内核提出了一个虚拟总线：platform总线，platform设备和platform驱动。

## 3.1、platform总线注册

platform\_bus\_init

-> bus\_register

注册的内容就是：

struct bus\_type platform\_bus\_type = {

**.name = "platform",**

.dev\_groups = platform\_dev\_groups,

.match = platform\_match,

.uevent = platform\_uevent,

.pm = &platform\_dev\_pm\_ops,

}

对于platform平台而言，platform\_match函数就是月老，负责驱动和设备的匹配。

## 3.2 platform驱动

结构体为platform\_driver，结构体内容为：

struct platform\_driver {

**int (\*****probe)(struct platform\_device \*);**

int (\*remove)(struct platform\_device \*);

void (\*shutdown)(struct platform\_device \*);

int (\*suspend)(struct platform\_device \*, pm\_message\_t state);

int (\*resume)(struct platform\_device \*);

**struct device\_driver driver;**

**->** **const struct of\_device\_id \*of\_match\_table;**

**-> const char \*name;**

**const struct platform\_device\_id \*id\_table;**

bool prevent\_deferred\_probe;

};

使用platform\_driver\_register向内核注册platform驱动。

platform\_driver\_register

-> \_\_platform\_driver\_register (platform\_driver)

-> 设置driver的probe为platform\_drv\_probe， //如果platform\_driver的

// probe函数有效的话。

-> driver\_register

->执行device\_drive->probe，对于platform总线，也就是platform\_drv\_probe函数。而platform\_drv\_probe函数会执行platform\_driver下的probe函数。

结论：向内核注册platform驱动的时候，如果驱动和设备匹配成功，最终会执行platform\_driver的probe函数。

## 3.3 platform设备

结构体platform\_device：

struct platform\_device {

const char \*name;

int id;

bool id\_auto;

struct device dev;

u32 num\_resources;

struct resource \*resource;

const struct platform\_device\_id \*id\_entry;

char \*driver\_override; /\* Driver name to force a match \*/

/\* MFD cell pointer \*/

struct mfd\_cell \*mfd\_cell;

/\* arch specific additions \*/

struct pdev\_archdata archdata;

};

1、无设备树的时候，此时需要驱动开发人员编写设备注册文件，使用platform\_device\_register函数注册设备。

2，有设备树，修改设备树的设备节点即可。

当设备与platform的驱动匹配以后，就会执行platform\_driver->probe函数。

## 3.4 platform匹配过程

根据前面的分析，驱动和设备匹配是通过bus->match函数，platform总线下的match函数就是：platform\_match。

platform\_match

-> of\_driver\_match\_device，设备树

-> acpi\_driver\_match\_device ACPI类型的

-> platform\_match\_id 根据platform\_driver-> id\_table

-> strcmp(pdev->name, drv->name) //最终的就是比较字符串，就是platform\_device->name，和platform\_driver->driver->name。无设备树情况下使用。

1、有设备树的时候：

of\_driver\_match\_device

-> of\_match\_device(drv->of\_match\_table, dev) //of\_match\_table非常重要，

类型为of\_device\_id。

//compatible属性

->

# 四、实验程序编写

## 4.1、无设备树

两部分：platform\_driver、platform\_device。以点灯为例。

1、编写向platform总线注册设备。编写驱动需要寄存器地址信息，地址信息使用设备信息，定义在platform\_device里面，因此需要在驱动里面获取设备中的信息，或者叫资源。使用函数platform\_get\_resource()

## 4.2、有设备树

有设备树的时候设备是由设备树描述的，因此不需要向总线注册设备，而是直接修改设备树。只需要修改设备树，然后编写驱动。

驱动和设备匹配成功以后，设备信息就会从设备树节点转为platform\_device结构体。

platform提供了很多API函数去获取设备相关信息的。

# 五 、运行测试